1. 上课约定须知

课程主题：SparkCore第一次课 -- Spark 架构核心设计
上课时间：20:00 - 23:00
课件休息：21:30 左右 休息10分钟
课前签到：如果能听见音乐，能看到画面，请在直播间扣 666 签到

2. 上次作业复盘

在 Hive 的课程中，给大家布置的作业有两份：

```
1、最终大作业：5个SQL面试题
2、影评案例：10个SQL编写题
```

请大家抽时间尽早完成，另外也请大家，在工作过程中，或者面试中碰到一些有意思的 SQL 题目，发给我。我整理成文档反馈给各位。

3. 上次内容总结

咱们从开课之初讲解了（HDFS，Kafka，ZooKeeper，HBase）之外，就开始讲解计算相关的技术组件，前面三次课是讲解关于离线数仓构建过程中，最重要，最常用的一个技术组件 Hive 的相关知识。主要讲解的内容是：

```
1、第一次课：Hive 企业应用案例 和 经典SQL 技巧总结
2、第二次课：Hive 30招 调优 + 两个经典案例
3、第三次课：Hive 源码解析：SQL如何转换成 MapReduce
```

总的来说，学会写 SQL 是为了应对工作中的开发任务，弄清楚底层的原理，是为了应对面试或者SQL的调优等相关。

4. 本次内容大纲

接下来，开始进入 Spark 的学习：Spark 的学习，主要分为两个方面，一方面是 架构设计，一方面是源码解析。总共四次课

```
1、Spark架构设计 -- Spark 核心功能
2、SparkCore源码分析 -- SparkRPC 和 集群启动
3、SparkCore源码分析 -- Spark App提交和 SparkContext初始化
4、SparkCore源码分析 -- Task 执行 和 Shuffle流程 源码剖析
```

今天是 SparkCore 的第一次课， 主要讲解的 Spark 的架构设计方面的知识点，包括以下这些：

```
1、Spark DAG 引擎剖析
2、Spark 核心功能和架构设计
3、Spark 运行机制和任务执行流程详解
4、Spark Shuffle详解
5、Spark 内存模型
```

5. 详细课堂内容

5.1. Spark DAG引擎剖析

5.1.1. MapReduce 的产生背景

Google 的 Nutch 爬虫爬取了海量的互联网网页数据，需要进行 倒排索引 和 PageRank 计算，那么需要把单机计算程序，扩展成分布式计算应用程序，但是会遇到非常多的问题：

```
1、数据存储的问题，首先需要搞定海量数据存储的问题。
2、运算逻辑至少要分为两个阶段，先并行计算（map），然后汇总（reduce）结果
3、这两个阶段的计算如何启动？如何协调？
4、运算程序到底怎么执行？数据找程序还是程序找数据？
5、如何分配两个阶段的多个运算任务？
6、如何管理任务的执行过程中间状态，如何容错？
7、如何监控和跟踪任务的执行？
8、出错如何处理？抛异常？重试？
```

可见在程序由单机版扩成分布式版时，会引入大量的复杂工作。为了提高开发效率，可以将分布式程序中的公共功能封装成框架，让开发人员可以将精力集中于业务逻辑。基于此，Google 研发了一个集资源管理和调度，应用程序编写框架于一体的分布式计算引擎 MapReduce，并与 2004 年发表论文公开 MapReduce 设计思想，但是并未开源。直到 2006年，NDFS 和 MapReduce 才从 Nutch 中独立出来并命名为 Hadoop，并于 2008 年成为 ASF 的顶级项目。

5.1.2. MapReduce 的缺点

MapReduce 的产生是一大进步，毋庸置疑，但是随着分布式计算引擎的快速发展，后续的分布式计算引擎诸如 Spark，Flink 等更符合企业生成需求。总结来说，MapReduce 的缺点如下：

```
1、MapReduce 的主节点 JobTracker 扩展性不足（MapReduce在2.x以前是一个集群，提供资源管理和编程API，主从架构）
2、MapReduce 的主节点 JobTracker 可用性差（没有 HA 机制）
3、MapReduce 集群的资源利用率低（关注 MapReduce-v2 以前的 slot 概念）
4、不能支持多种分布式计算应用程序框架（Tez, Storm, Spark, Flink等）
5、基于磁盘 IO 的设计导致程序运行过程中的频繁磁盘读写成为系统的性能瓶颈。只适合离线批处理。
```

需要注意的是：前四个缺点，已经在 Hadoop-2.x 版本中，都已经解决了。

Hadoop: 分布式操作系统（YARN） + 分布式文件系统（HDFS）+ 分布式计算应用程序编写框架（MapReduce）

事实上，现在的 Hdaoop，你认为可以认为它是分布式操作系统，因为他最强大的能力就是把多台物理上分散独立的服务器组织成一个整体，制定了资源管理规范（YARN），提供了一个文件系统（HDFS），也给出了分布式应用程序的编写规范（MapReduce）。

5.1.3. Spark 的优势

Spark 的优势总结如下：

```
01、减少磁盘IO（spark允许中间输出和结果存储在内存中，从而避免大量磁盘IO）
02、增加并行度（多个Stage可并行执行，多个MapReduce串联执行，mapper reducer mapper reducer .....）
03、避免重新计算（如果某个Task执行失败，会重新调度该Stage，但是会过滤已经成功的Task，避免重复计算）
04、可选的Shuffle排序（MapReduce一定会按照key排序，而Spark可选择是否排序以及是Map端排序，还是Reduce端排序）
05、灵活的内存管理策略（引入堆外内存，减少GC，软化存储内存和执行内存的边界，提高内存资源的资利用率）
06、易使用（多种编程语言支持和很多的高级操作运算符支持）
07、支持交互式（借助Scala的Iloop库提供REPL(Read Eval Print Loop)支持）
08、checkpoint支持（Spark的linage支持从某个失败的RDD的父RDD开始重新构建，而不用全部重来）
09、支持SQL（MapReduce依赖Hive等组件提供SQL支持）
10、支持流式开发（MapReduce根本不支持流式开发，Spark是一个一站式的分布式计算引擎）
11、可用性高（支持cluster的高可用，和应用程序application的高可用）
12、丰富的数据源支持（支持除HDFS之外的各种常用数据源，比如HBase，MySQL，Kafka，Alluxio等）
13、丰富的文件格式支持（支持普通文本格式，支持CSV，支持JSON，支持ORC，支持Partquet格式等）
```

基于以上的这些优势，Spark的官网总结宣称道：Spark运行效率 和 MapReduce 的性能比：100 : 0.9，其实这并不是一个公平的比较，因为这是在执行逻辑回归的场景中比较出来的，在这多次迭代的计算场景中，Spark 尤其能发挥优势，但是如果就只是运行一个简单的分布式计算程序(只有一次 shuffle)：10-3 ： 1，大概也就是一个个位数的优势比。

在鄙人看来，其实 Spark 真正的优势之处在于：Spark 简化了用户复杂应用程序的开发（非一次 Shuffle的复杂计算逻辑，内部的 DAG 执行引擎充分发挥作用，整合 RDD 数据可以持久化到内存的功能），整合各种数据源，提供了各种高阶计算算子。

既然这样，我们先来说一说，Spark 到底是怎么简化应用程序的开发的。然后再跟各位讲解一下 Spark 的 DAG 执行引擎（这种思想也应用在了 Flink 之中）。

5.1.4. Spark 编程套路

Spark 不仅仅是在内部执行引擎中相较于 MapReduce 有了一个很大的提升。并且，给用户提供的使用接口也有了相应的优化，在使用体验上，比 MapReduce 要很多很多。总结来说：**Spark 执行引擎，不管数据在哪里，不管数据集是什么格式，不管数据集有多大，不管用户执行什么逻辑的计算，Spark 执行引擎都能解决。**关于 Spark 对应用程序的抽象，主要分为三类：

```
1、编程执行入口，这个概念就跟传统意义上的上下文对象有一样的作用，称之为 XXXContext
2、数据集抽象，Spark针对不同的数据处理模式，提供了不同场景中的不同作用的数据数据集抽象，这是相对于mapreduce的一个极大进步
3、操作算子，mapreduce编程框架只提供了两个操作算子map和reduce，这就是得用户如果进行复杂计算，就需要定义多个mapreduce来进行串联执行，这是极为耗时的操作，但是Spark不一样，提供了80多个高阶计算算子，方便用户通过这些算子，完成各种负责计算逻辑的组装。
```

正因为如此高级的抽象封装，使得 Spark 应用程序的编写，使用就是一个简单的套路，总结如下：

```
1、获取一个编程入口 SparkContext
    val conf = new SparkConf().setAppName("WordCount");
    val sparkContext:SparkContext = new SparkContext(conf);

2、通过编程入口对象获取数据抽象：RDD（分布式计算引擎的数据必然来源于分布式存储系统）
    待计算的数据一定是分布式存储的（存在很多个节点联合存储这一份数据）
    RDD：不可变的容错的弹性的分布式数据集合（分布式集合）
    val dataRDD: RDD[String] = sparkContext.textFile("hfds_path");
```

```
3、调用RDD之上的各种逻辑操作算子执行数据计算：dataRDD.xx1().xx2().xx3().xx4()...
    针对这种分布式集合，有可能执行的各种操作都封装成算子
    rdd2 = rdd.xx1();
    rdd3 = rdd2.xx2();
    ...
    resultRDD = rddn.xxn();

4、任务提交
    resultRDD.action();

5、关闭SparkContext
    sparkContext.stop()
```

通过以上程序的编写逻辑，可以总结出来了一个复杂的分布式计算任务的多个 RDD 之间是有依赖关系的!

设计：

```
1、下游RDD与上游RDD之间的分区依赖有可能是 1对1（length），也有可能是 1对多（sum）

2、把多个连续的1对1关系的RDD，放在一个阶段，以管道方式来执行。这种思想，其实在 MapReduce 也能体现出来，比如以下这段代码的执行：word.lower().filter().length()

3、既然多个1对1的RDD放在一起形成一个阶段，那么多对1的依赖，就会被切开，上游的所有Task必须执行完毕之后，下游才能执行。这样子的话，不管多复杂的计算逻辑，通过 RDD 之间的调用和依赖关系，就可以被划分成多个不同的执行阶段。
```

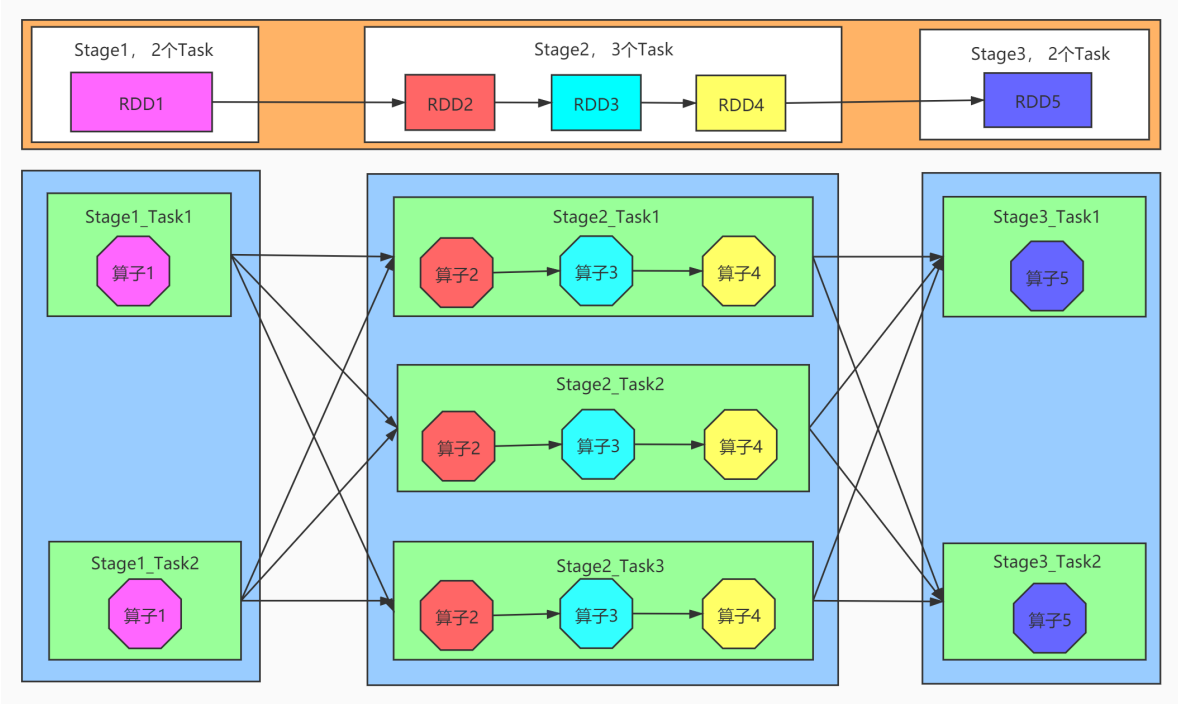
这种思路的总结，就是另种一种通用的执行机制：DAG 执行引擎，简而言之：DAG 执行引擎就是把 分布式计算应用程序看做是 多阶段的分布式并行执行框架。

补充一点：学习分布式计算引擎的难点，就在于上一个阶段与下一个阶段直接的 数据混洗过程。这个过程尤其复杂也容易出问题。

5.1.5. DAG 引擎剖析

DAG 是有向无环图，一般用来描述任务之间的先后关系， Spark 中的 DAG 就是 RDD 内部的转换关系，这些转换关系会被转换成依赖关系，进而被划分成不同阶段，从而描绘出任务的先后顺序。

```
DAGscheduler 在 Spark 引擎中的作用就是负责任务的逻辑调度，负责将job作业拆分成不同阶段的具有依赖关系的多批次任务，并且制定调度逻辑。
```



DAG 执行引擎的显著优点：

```
1、提高 Scheduler 的调度效率（多个没关系的stage的调度可以并行执行）
2、支持基于 Linage 的 Fault Tolerance
3、有向无环，无循环依赖
4、减少中间结果的落盘和读取操作次数从而提高任务执行效率，mapreduce性能差的原因就在于执行过程中的任何的shuffle read和write都是基于磁盘。Spark 提供了基于内存的存储体系功能的支撑：支持我们把数据持久化到内存，方便下一个阶段的Task来获取
```

其实，MapReduce 作为第一代分布式计算引擎，多少也能体现出 DAG 的影子，MapReduce 只把分布式计算，归结为 Map 和 Reduce 两种操作，但是复杂的计算，可能就需要多个这样的步骤。如果你真的为了实现一个需求计算而编写了多个串联的 MapReduce 程序来执行，那么 MapReduce 实际上也是形成了 DAG 计算的。所以，其实你可以认为：Spark 中的一个 两个 stage 的计算和他们之间的 shuffle，就是一个 MapReduce 程序。Spark 引擎的优势是天然把复杂的分布式计算过程进行了完整抽象，使用户只用关注每个 Stage 的计算逻辑，而不是关注 Stage 与 Stage 之间的依赖逻辑。

流式处理 和 批处理的最大不同是什么（小补充知识：都会分成多个阶段 stage 来执行）：

```
1、批处理是 上一个stage 调度执行完毕之后，再去调度下一个stage执行，task 的执行是有结束的
2、流式处理 所有stage 中的所有 task 同时调度执行，除非有明确的结束命令，否则所有Task都会一直运行
```

造成这个不同的最重要的原因，就在于 数据是否是连续不断的产生，还是说，数据集有一个明确的结束点！

5.2. Spark 核心功能和架构设计

5.2.1. Spark 网络通信框架 RPC

分布式系统必备的基础组件之一，就是分布式网络通信框架。

在 Spark 的内部的很多地方都涉及网络通信，比如 Spark 各个组件之间的互相通信，用户文件与 Jar 包的上传，节点之间的 Shuffle 过程，Block 数据的复制和备份。在 Spark 0.x 和 Spark 1.x 版本中，使用 Akka 轻松构建强有力的高并发和分布式网络通信，但是在 Spark 2.x 被移除了。Spark官方对此的描述是："Akka的依赖被移除了，因此用户可以使用任何版本的 Akka 来进行编程了。"

```
1、Spark-1.x 中，用户文件和 jar 包上传，采用 jetty 实现的 HttpFileServer 实现的，但是 Spark-2.x 废弃了，现在使用基于 Spark 内置 RPC 框架 NettyStreamManager

2、Shuffle 过程 和 Block 数据复制和备份在 Spark-2.x 版本依然沿用 Netty，通过对接口和程序的重新设计，将各个组件间的消息互通，用户文件和 jar 包的上传也一并纳入 Spark 的 RPC 框架。
```

Spark 的 RPC 组件的大致描述：

```
1、TransportContext 内部包含：TransportConf 和 RpcHandler
2、TransportConf 在创建 TransportClientFactory 和 TransportServer 都是必须的
3、RpcHandler 只用于创建 TransportServer
4、TransportClientFactory 是 Spark RPC 的客户端工厂类实现
5、TransportServer 是 Spark RPC 的服务端实现
6、TransportClient 是 Spark RPC 的客户端实现
7、ClientPool 是 TransportClientFactory 的内部组件，维护 TransportClient 池
8、TransportServerBootStrap 是 Spark RPC 服务端 引导程序
9、TransportClientBootstrap 是 Spark RPC 客户端 引导程序
10、MessageEncoder 和 MessageDecoder 编解码器
```

关于 Spark RPC 的案例，会在讲 SparkCore 源码的时候进行讲解！

总结：

```
1、Spark RPC 客户端的具体实现是 TransportClient，由 TransportClientFactory 创建，TransportClientFactory 在实例化的时候需要 TransportConf，创建好了之后，需要通过 TransportClientBootstrap 引导启动，创建好的 TransportClient 都会被维护在 TransportClientFactory 的 ClientPool 中。

2、Spark RPC 服务端的具体实现是 TransportServer，创建的时候，需要 TransportContext 中的 TransportConf 和 RpcHandler。在 init 初始化的时候，由 TransportServerBootStrap 引导启动。

3、在 Spark RPC 过程中，实现具体的编解码动作的是： MessageEncoder 和 MessageDecoder。可以集成多种不同的序列化机制。
```

5.2.2. Spark 序列化

大数据场景中常见的三种序列化机制：

```
1、Java的原生序列化机制：Serializable

2、ZooKeeper 和 Hadoop：自行设计实现：自定义的类实现：带有序列化和反序列化抽象方法的接口
    Hadoop实现：class Student implements Writable
    ZooKeeper实现：class Student implements Record

3、Spark的序列化，提供了两种序列化支持：
    1、默认实现：Java的原生序列化机制：JavaSerializable
    2、Kryo机制：KyroSerializable
```

```
声明去启用，并且参与序列化的类，必须要注册！
支持三种类型：
1、各种基本类型
2、String类型
3、基本类型的数组类型
```

套用官文 Tuning Spark 中的一句话作为文章的标题：

```
Often, choose a serialization type will be the first thing you should tune to
optimize a Spark application.
```

在 Spark 的架构中，在网络中传递的或者缓存在内存、硬盘中的对象需要进行序列化操作，序列化的作用主要是利用时间换空间：

```
1、分发给 Executor 上的 Task
2、需要缓存的 RDD（前提是使用序列化方式缓存）
3、广播变量
4、Shuffle 过程中的数据缓存
5、使用 receiver 方式接收的流数据缓存
6、算子函数中使用的外部变量
```

上面的六种数据，通过 Java 序列化（默认的序列化方式）形成一个二进制字节数组，大大减少了数据在内存、硬盘中占用的空间，减少了网络数据传输的开销，并且可以精确的推测内存使用情况，也可以降低 GC 频率（对象在堆中是离散的，序列化之后存储在堆中是连续的）。

SerializerManager 提供了 getSerializer 接口， 会自动选择选用哪种序列化方式， 默认为 Java 自带的序列化。我们只需要调用 SerializerManager 的相关方法，就可很方便的序列化数据。
SerializerManager 统一了 Java序列化 和 Kryo 序列化的接口。目前 KryoSerializer，支持序列化的数据类型，有字符串类型，基本数据类型和其对应的数组类型这几种。

官文介绍，Kryo 序列化机制比 Java 序列化机制性能提高 10 倍左右，Spark 之所以没有默认使用 Kryo 作为序列化类库，是因为它不支持所有对象的序列化，同时 Kryo 需要用户在使用前注册需要序列化的类型，不够方便。

Kryo 需要用户进行注册，这也是为什么 Kryo 不能成为 Spark 序列化默认方式的唯一原因，但是建议对于任何 “网络密集型”(network-intensive) 的应用，都采用这种方式进行序列化方式。

```scala
// 创建SparkConf对象
val conf = new SparkConf().setMaster(...).setAppName(...)
// 设置序列化器为KryoSerializer
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
// 注册要序列化的自定义类型
conf.registerKryoClasses(Array(classOf[MyClass1], classOf[MyClass2]))
```

5.2.3. SparkEnv

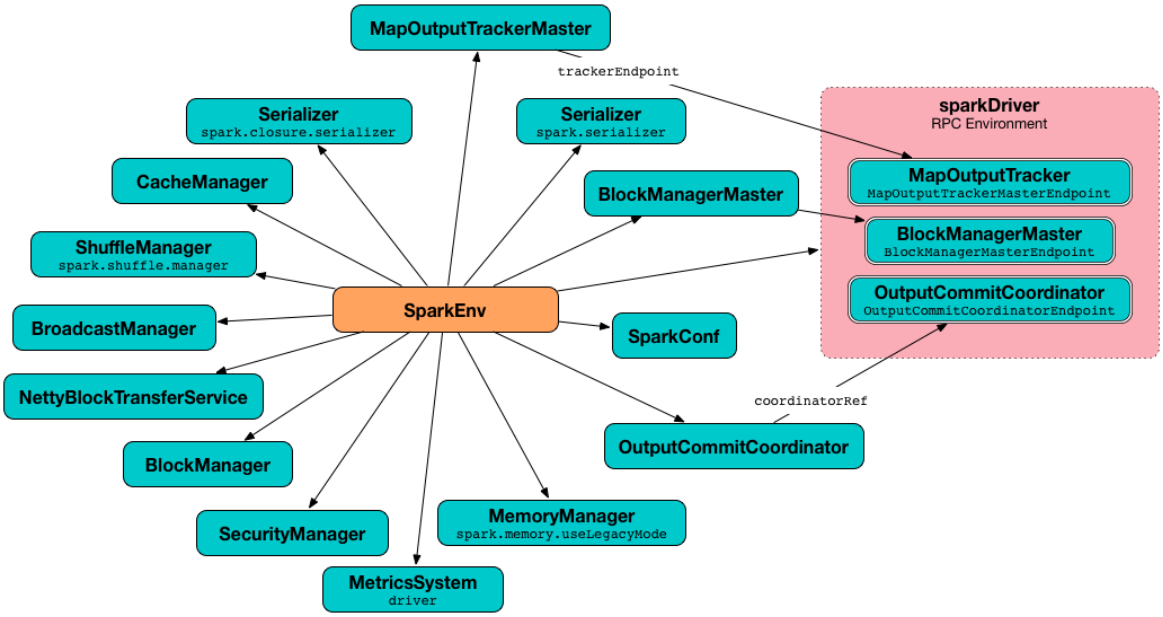
你要学好 Spark，彻底掌握 Spark 的机制，必须掌握三个对象：

```
1、SparkConf          管理各种配置（通过内部的 ConcurrentHashMap 实现），作用等同于 Hadoop 的 configuration
2、SparkEnv           内置和管理各种子系统实例对象，比如 MemoryManager，SerializerManager，ShuffleManager 等
3、SparkContext       管理了 SparkConf 和 SparkEnv，是 Spark 应用程序的编程入口
```

Spark 执行环境 SparkEnv 是 Spark 应用程序中的 Task 运行所必须的各种组件的容器，SparkEnv 内部封装了 RPC 环境（RpcEnv），序列化管理器，广播管理器，Map任务输出跟踪器，存储体系，度量系统，输出提交协调器等 Task 运行所需的各种组件。

```
01、RpcEnv
02、SerializerManager
03、MapOutputTracker
04、ShuffleManager
05、BroadcastManager
06、BlockManager
07、SecurityManager
08、MetricsSystem
09、MemoryManager
10、OutputCommitCoordinator
11、SparkConf
```

借用网络中的一张图：



5.2.4. Spark 调度系统

5.2.4.1. RDD 和 DAG引擎 和 Stage

相关核心知识点：

1、RDD学名：弹性分布式数据集，是一个容错的并行的数据结构，可以控制将数据存储到磁盘或内存，能够获取数据的分区等，RDD还提供了一堆用于计算的类似于Scala函数的高阶操作算子用于完成计算。

2、RDD按照依赖关系，会构建成有向无环图，既DAG，RDD之间的关系有两种：NarrowDependecy和ShuffleDependency，NarrowDependecy会被划分到同一个Stage中，以管道的方式迭代执行。ShuffleDependency 由于所依赖的 分区Task 不止上游一个 Task，往往需要跨节点传输。

3、RDD本身是一个不可变的分布式数据集，NarrowDependecy只需要重新执行父RDD的丢失分区的计算既可以恢复，但是ShuffleDependecy则需要考虑恢复所有父RDD的丢失分区。失败的Task重新执行，执行成功的Task不执行，从Checkpoint中进行读取即可。

4、RDD分区，由于RDD是一个分布式集合，则可以进行并行计算提高效率。RDD的分区可以根据业务需求和硬件资源来进行并行任务的数量的控制，从而提高任务的执行效率。

5、Spark的Application会按照ShuffleDependency划分成多个Stage，最后一个Stage称之为ResultStage，前面的Stage称之为ShuffleMapStage，Stage和Stage之间，需要进行数据Shuffle操作，可能导致数据倾斜。

6、一个stage中到底有多少个task：由当前这个stage的最后一个RDD的分区个数来决定

```
DAG
    DAGScheduler 负责把 构建出来的 DAG 切分成多个stage  切分标准：shffle算子 ShuffleDenpedency
    RDD与RDD之间的转换关系依赖：
        宽依赖 ShuffleDenpedency
        窄依赖 NarrowDependency
Stage
    一个stage中，其实可能包含多个RDD，和多个计算逻辑，
    但是没关系，上下RDD之间的依赖关系就只是普通的一一对应的关系，RDD与RDD之间就是窄依赖
    在一个stage之间，所有的RDD的依赖都是窄依赖。
    上一个stage的最后一个 RDD 和下一个stage的第一个RDD之间是宽依赖
```

5.2.4.2. 面向 DAG 的 DAGScheduler

逻辑调度器

DAGScheduler 负责把 构建出来的 DAG 切分成多个 stage，切分标准：shffle算子 (ShuffleDenpedency)

1、主要作用：维护active jobs队列，维护waiting stages、active stages和failed stages三个stage队列，以及job和stages之间的映射关系

2、工作机制：DAGScheduler拿到一个Job，会切分成多个Stage：从job的算子调用中从后面往前寻找shuffle算子，如果找到一个shuffle算子，就切开，已经找到的RDD的执行链就自成一个Stage，放入到一个栈中。将来DAGScheduler要把这个栈中的每个stage拿出来，提交给TaskScheduler来调度执行。

```
3、核心描述
    一个Application = 多个job
    一个job = 多个stage，也可以说一个application = 多个stage
    一个Stage = 多个同种task并行运行
    Task 分为 ShuffleMapTask 和 ResultTask，具体为：ShuffleMapTask1 --> ShuffleMapTask2 --> ShuffleMapTask3 --> ResultTask
    Dependency 分为 ShuffleDependency宽依赖 和 NarrowDependency窄依赖，DAGScheduler执行 stage的切分，切分依据为宽依赖
```

5.2.4.3. 任务调度器 TaskScheduler

物理调度器：接收一个 Stage，然后解析封装成多个 LaunchTask 对象，发送给已经启动好了的存在于 Worker 节点的 Executor 中来执行。真正启动的 Task 有两种类型：ShuffleMapTask + ResultTask

```
1、TaskScheduler 本身是个特质，Spark 里只实现了一个 TaskSchedulerImpl，理论上任务调度可以定制。

2、维护 task 和 executor 对应关系，executor 和物理资源对应关系，在排队的 task 和正在跑的 task。

3、维护内部一个任务队列，根据 FIFO 或 Fair 策略，调度任务。

4、TaskScheduler 的作用：把 Stage 变成 TaskSet，然后执行任务分发。有两个重要的成员变量：
    1、DAGScheduler：负责JOb中的stage切分，
    2、SchedulerBackend：执行Task的分发
```

总结一下： 真正执行 LaunchTask 消息发送的是 TaskScheduler 的内部成员变量：SchedulerBackend，同样的，在 Executor 的内部，也会启动一个 ExecutorBackend 来接收这个消息，并且提交任务到 Executor 的线程池来执行。！

5.2.4.4. SchedulerBackend 和 ExecutorBackend

一个 Spark 应用程序在执行的时候，其实是由一个 Driver 主控，一堆 Executor 来负责执行 Task，他们都是 JVM 进程。那么他们之间的交互是如何实现的呢?

SchedulerBackend 存在于 Driver，负责进行调度和 Task 派发
ExecutorBackend 存在于 Executor，负责接收 Task派发，和执行状态反馈

```
1、在 TaskScheduler 下层，用于对接不同的资源管理系统，提供了一个接口：SchedulerBackend，需要实现的主要方法有：
    def start():Unit
    def stop():Unit
    def reviveOffers():Unit  // 申请资源
    def killTask(taskId: Long, executorId: String, ...): Unit = throw new UnsupportedOperationException

2、ExecutorBackend 存在于 Executor 中，用来和 SchedulerBackend 进行通信。
```

5.2.4.5. SparkContext 应用程序执行上下文

```
1、SparkContext 是用户通往 Spark 集群的唯一入口，可以用来在 Spark 集群中创建 RDD、累加器 Accumulator 和广播变量 Braodcast Variable，如果你想要什么功能或者什么组件，或者什么配置，你问他就行。Spark 最重要的三大抽象：SparkContext,RDD,算子

2、SparkContext 也是整个 Spark 应用程序中至关重要的一个对象，可以说是整个应用程序运行调度的核心(不是指资源调度)

3、SparkContext 存在于 Driver端，在实例化的过程中会初始化 DAGScheduler、TaskScheduler 和 SchedulerBackend

4、SparkContext 会调用 DAGScheduler 将整个 Job 划分成几个小的阶段(Stage)，TaskScheduler 会调度每个 Stage 的任务(Task)执行处理。
```

5.2.5. Spark 配置管理

SparkConf 是 Spark 的配置管理类，每一个 Spark 组件都直接或者间接的使用它存储的属性。这些属性都被存储在：

```
private val settings = new ConcurrentHashMap[String, String]()
```

记住：key 和 value 都是 String 类型，为了通用!

SparkConf 可以通过三种方式获取：

```
1、系统参数，在 SparkConf 初始化的时候通过：System.getProperties() 获取以 spark. 作为前缀的属性。
2、使用 SparkConf 的 API 进行设置：sparkConf.set(key,value)
3、从其他 SparkConf 克隆，比如初始化 SparkContext 的时候
```

HDFS 中的 配置的管理是怎么做的？到底什么时候加载的 core-site.xml 和 hdfs-site.xml。

那，我们到底可以通过哪些方式来设置参数呢?

```
1、全局设置：集群配置文件
2、应用程序设置：shell 命令设置，spark-submit
3、程序代码中设置：sparkConf.set(key, value)
```

5.2.6. Spark API设计

Spark 编程套路： DAG 引擎的优秀设计：SparkContext + RDD + 算子 （提供了大量的简单易用的常用算子，也提供了自定义）

```
1、source       加载数据源，支持自定义数据源
2、transform    算子（ RDD --计算（转换）--> RDD）
3、sink         输出数据结果
```

```
1、获取编程入口                          环境对象 链接对象
2、通过编程入口加载数据源获取数据抽象      Source 对象
3、针对数据抽象对象执行各种计算           Action Transformation
4、执行action算子触发任务提交运行         Submit & Sink
5、关闭编程入口                          stop close
```

由于 Spark 是以一个一站式的 分布式计算引擎，针对该引擎之上，提供了 SparkSQL，SparkStreaming 等高级功能，针对不同的引擎，提供了不同的编程入口和数据抽象：

1、编程模型：Spark-1.x 版本

	SparkCore	SparkSQL	SparkStreaming
编程入口	SparkContext	SQLContext + HiveCotext	StreamingContext
数据抽象	RDD	DataFrame	DStream

2、编程模型： Spark-2.x 版本， 原则是为了统一：

```
数据抽象：DataSet，原有的这种组件都转为幕后
编程入口：SparkSession
```

	SparkCore	SparkSQL	SparkStreaming
编程入口	SparkContext/SparkSession	SparkSession	StreamingContext
数据抽象	RDD / DataSet	DataFrame / DataSet	DStream

3、Spark Operator

```
Spark offers over 80 high-level operators that make it easy to build parallel
apps. And you can use it interactively from the Scala, Python, R, and SQL
shells.
```

http://spark.apache.org/docs/2.4.7/rdd-programming-guide.html#transformations
http://spark.apache.org/docs/2.4.7/rdd-programming-guide.html#actions

更多算子学习使用：http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html

一句话总结 Spark，相当的强大：

```
Spark 不管从哪里读取数据，不管读取的数据多大，是什么格式，以何种形式存在，Spark都可以读取，执行各种类型的计算，然后可以输出结果到各种数据目的地。
```

5.2.7. Spark 核心概念

注意几对核心概念，清楚的了解这些概念，是了解 Spark 底层运行原理的重要组成部分。

```
1、Master + Worker
    spark的standalone集群的主从节点的角色名称

2、Master(老板) + Driver(项目leader) + Executor(项目成员) + client(客户)
```

```
    用户编写的应用程序application分布式运行时的的概念：主从架构
    driver: 负责任务的调度和监控
    executor: 负责任务的执行
    用户提交Application一个，就会启动一个Driver + N个 Executor

3、Application + Job + Stage + Task
    Application：完整的用来实现某个业务的一个完成的程序：一个jar包中的带main方法的一个类
    Job：由这个application中的action算子来决定到底有多少个job，一个action会生成一个job
    Stage：一个job按照shuffle依赖切分成多个stage
    Task：stage中的可以并行运行的任务
```

除了 Task 在 Exeuctor 运行以外，其他的事情，都是 Driver 中完成。

总结：

```
1、在 spark application main 方法中的所有代码，都是在 driver 中执行的
2、你针对某些像 map reduce 等高阶算子传入了一个函数参数，这个函数参数，就是在 Executor 中执行的。
```

如果你在 Dirver 中定义了某个变量，但是在某个高结算子的 函数参数中，使用了，那么，这个变量就要从 Driver 序列化传送到 Executor 端！

```
val a:Int = 100
rdd.map(x => x + a)
```

这里有一个可调优的地方， 如果这个 a 变量很大，则一定要进行广播来实现！ 如果使用了广播机制，这个a变量只会被发送到 executor 中一次，但是如果没有使用广播变量，那么这个 a 变量就会被序列化到每个 Task 中一次。一个 Executor 事实上是并发运行多个 Task 的

5.2.8. Spark 引擎其他功能模块

Spark 执行引擎除了上面提到的这些核心功能之外，还有一些其他的辅助服务，这里呢，给大家提到，虽然不再详细讲解，但是也请大家引起注意。

```
1、Spark 事件总线
2、Spark 度量体系
3、Spark 存储体系
4、Spark 持久化体系
5、Spark 排序器
6、Spark Tungsten
7、Spark 文件服务
8、Spark Web UI
......
```

5.3. Spark 运行机制和任务执行流程详解

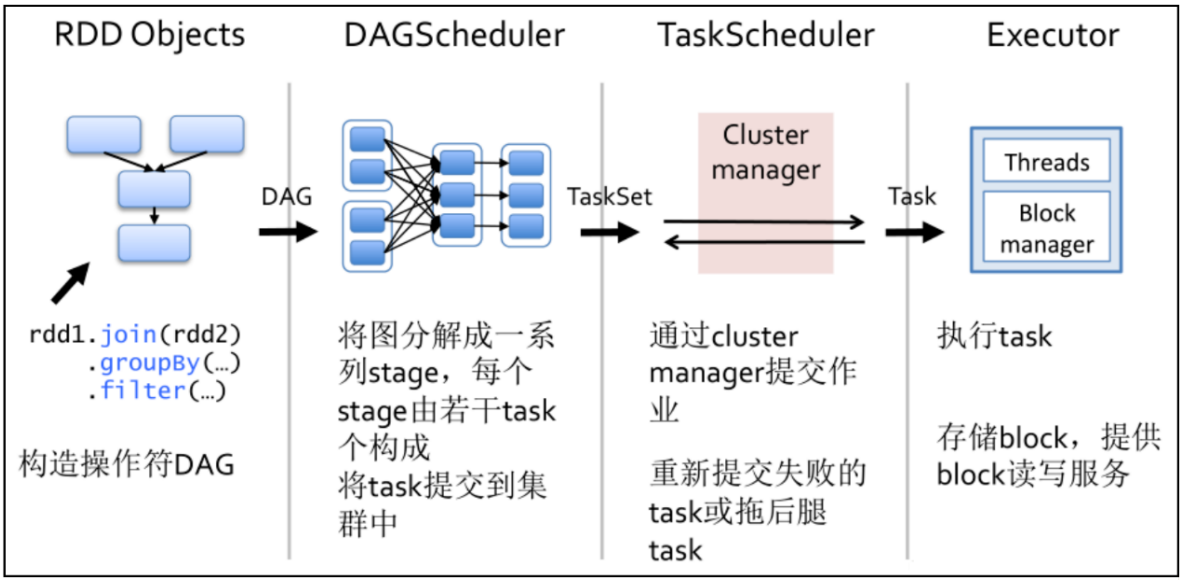
核心四大步骤：

1、构建DAG（driver当中完成）
使用算子操作RDD进行各种transformation操作，最后通过action操作触发Spark作业运行。提交之后Spark会根据转换过程所产生的RDD之间的依赖关系构建有向无环图。

2、DAG切割（driver当中完成）
DAG切割主要根据RDD的依赖是否为宽依赖来决定切割节点，当遇到宽依赖就将任务划分为一个新的调度阶段(Stage)。每个Stage中包含一个或多个Task。这些Task将形成任务集（TaskSet），提交给底层调度器进行调度运行。

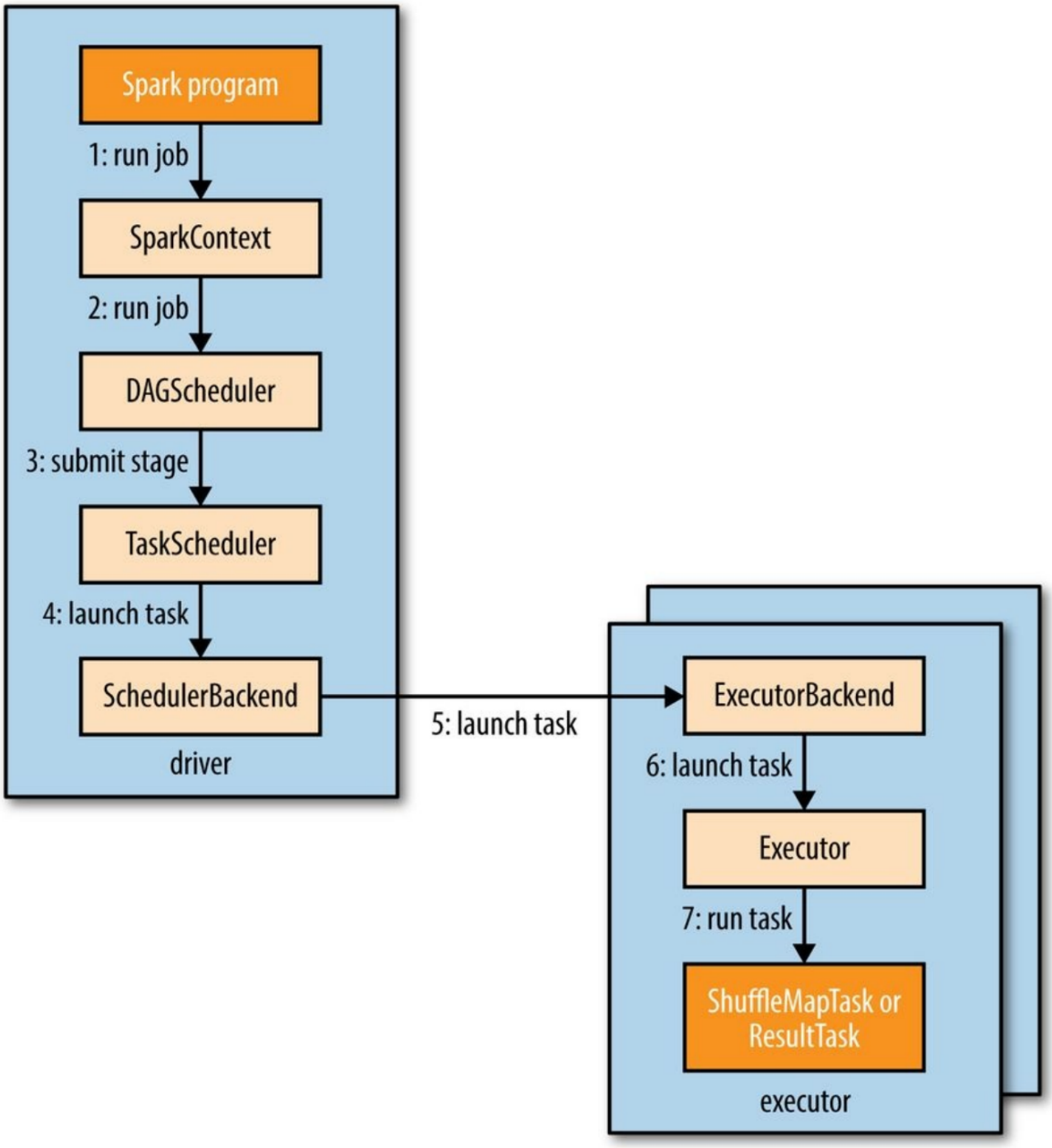
3、任务调度（driver当中完成）
每一个Spark任务调度器只为一个SparkContext实例服务。当任务调度器收到任务集后负责把任务集以Task任务的形式分发至Worker节点的Executor进程中执行，如果某个任务失败，任务调度器负责重新分配该任务的计算。

4、执行任务（worker中的exeuctor执行）
当Executor收到发送过来的任务后，将以多线程（会在启动executor的时候就初始化好了一个线程池）的方式执行任务的计算，每个线程负责一个任务，任务结束后会根据任务的类型选择相应的返回方式将结果返回给任务调度器。



Spark任务的核心执行流程主要分为四大步骤：换一种方式总结

```
Driver工作：Build DAG
DAGScheduler工作：Split DAG to Stage
TaskScheduler工作：Change Stage to TaskSet And Submit
Worker工作：execute Task
```



首先，用户编写好写应用程序，之后，都会达成jar包，通过 spark-submit 来提交。最终转交给 SparkSubmit.class ，通过提交模式可以找到对应的 客户端启动类。这个客户端类启动好了之后，执行一些参数的解析，执行jar包处理的等相关准备动作之后，就发送请求（ApplicationRegistrion）给 对应的 资源调度系统的主节点 Master。 Master 首先找到一个空闲的节点，来启动一个 Driver（Master 发送消息 LauchDriver 给 Worker），启动 Driver（启动Driver之后，马上有一个动作，就是启动这个 Job 需要的都少个 Executor），解析和执行 用户编写的应用程序的 main 方法。首先执行的是 SparkContext 的初始化。 然后执行各种操作算子，构建 DAG，最后触发 actioin 提交 job

```
1、应用程序在driver中执行
2、driver中执行应用程序的第一件事，就是初始化 SparkContext
3、在初始化SparkContext，会初始化三大组件 DAGScheduler，TaskScheduler，
SchedulerBackEnd
4、在初始化SchedulerBackend的时候，会初始化两个通信组件：ClientEndpoint +
DriverEndpoint
    ClientEndpoint 跟 Master 打交道，负责任务提交和 Driver 状态汇报
    DriverEndpoint 跟 Worker 中的 executor 打交道，负责任务派发和跟踪 Task 执行状态
5、DriverEndpoint 会发送 LaunchTask 命令给 ExecutorBackend
6、ExecutorBackend 接收到命令之后，会调用 launchTask 方法来启动任务，其实就是封装一个
TaskRunner的线程对象，提交给 Executor 的线程池来执行
7、整个过程，Executor都会维持跟 Driver 的心跳，整个过程中，Driver也会维持和 master的心跳
```

5.4. Spark Shuffle详解

学习分布式计算引擎，或者编写分布式计算应用程序，你必须了解 Shuffle！分布式计算引擎的设计难点是 Shuffle，同时 Spark 应用程序出问题最集中的地方也是 Shuffle。

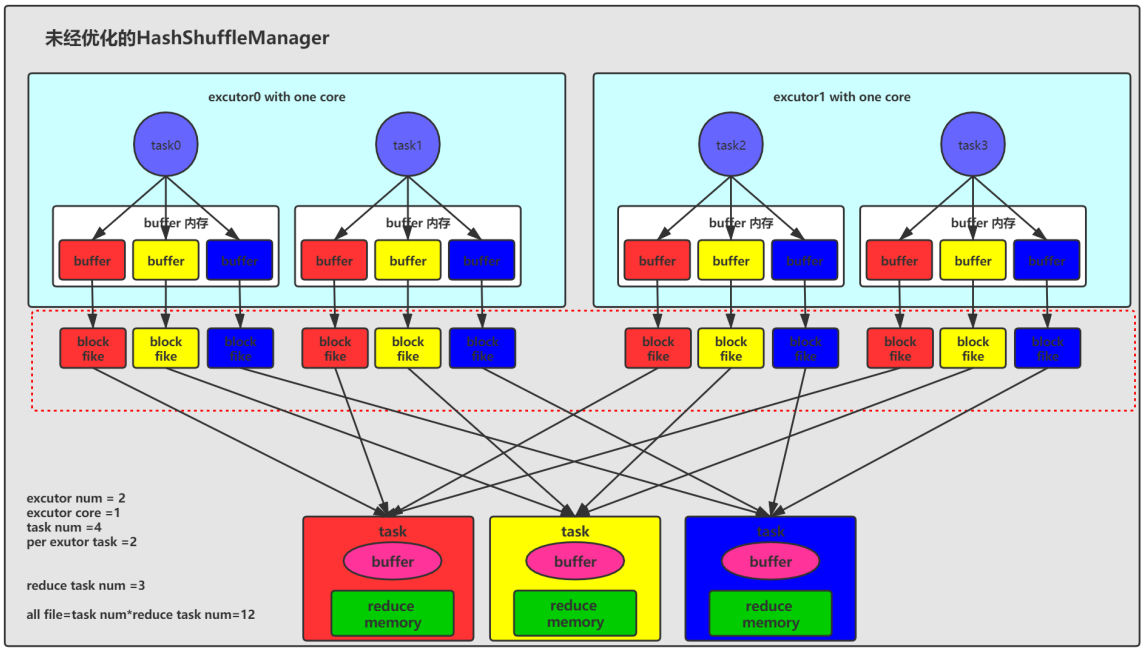
在 Spark 的源码中，负责 shuffle 过程的执行、计算和处理的组件主要就是 ShuffleManager，也即 shuffle 管理器。随着 Spark 的版本的发展，ShuffleManager 也在不断迭代，变得越来越先进。

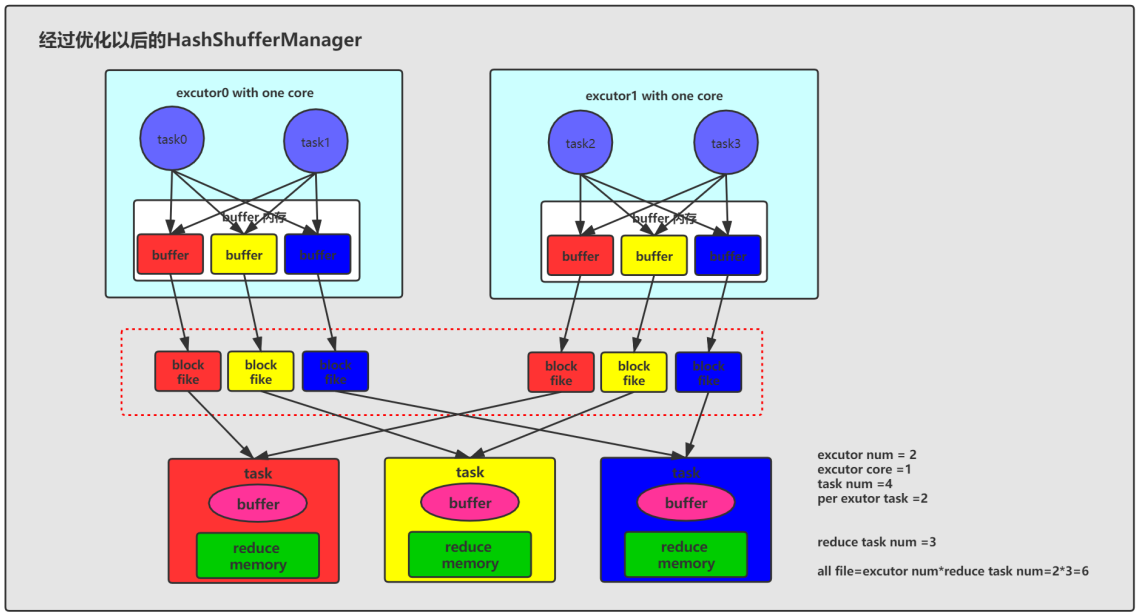
在 Spark 1.2 以前，默认的shuffle计算引擎是 HashShuffleManager。该 ShuffleManager 而 HashShuffleManager 有着一个非常严重的弊端，就是会产生大量的中间磁盘文件，进而由大量的磁盘 IO 操作影响了性能。

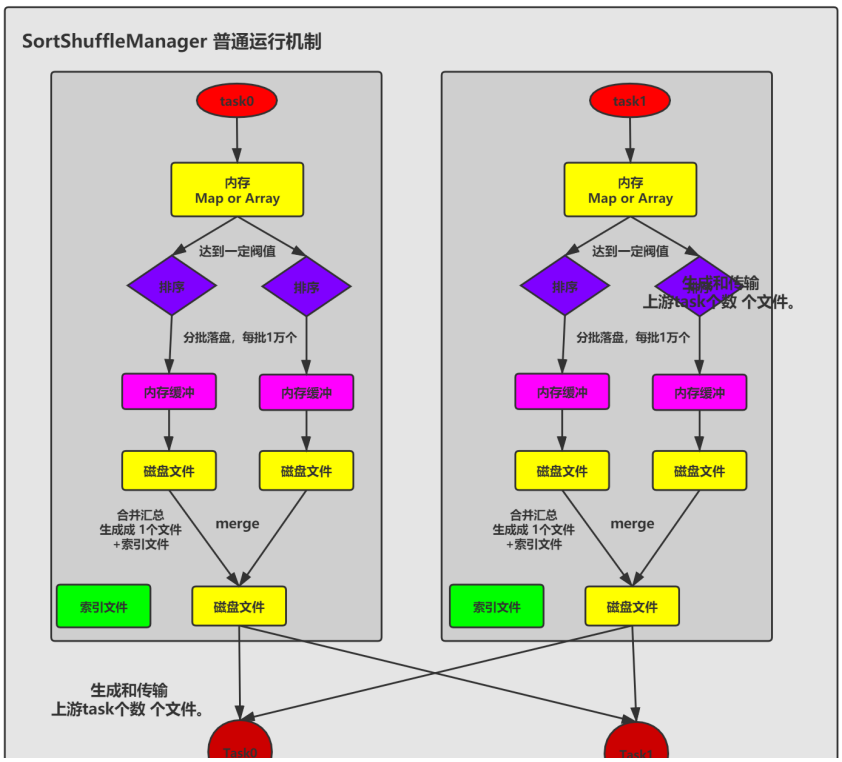
因此在Spark 1.2 以后的版本中，默认的 ShuffleManager 改成了 SortShuffleManager。SortShuffleManager 相较于 HashShuffleManager 来说，有了一定的改进。主要就在于，每个 Task 在进行 shuffle 操作时，虽然也会产生较多的临时磁盘文件，但是最后会将所有的临时文件合并（merge）成一个磁盘文件，因此每个 Task 就只有一个磁盘文件。在下一个 stage 的 shuffle read task 拉取自己的数据时，只要根据索引读取每个磁盘文件中的部分数据即可。

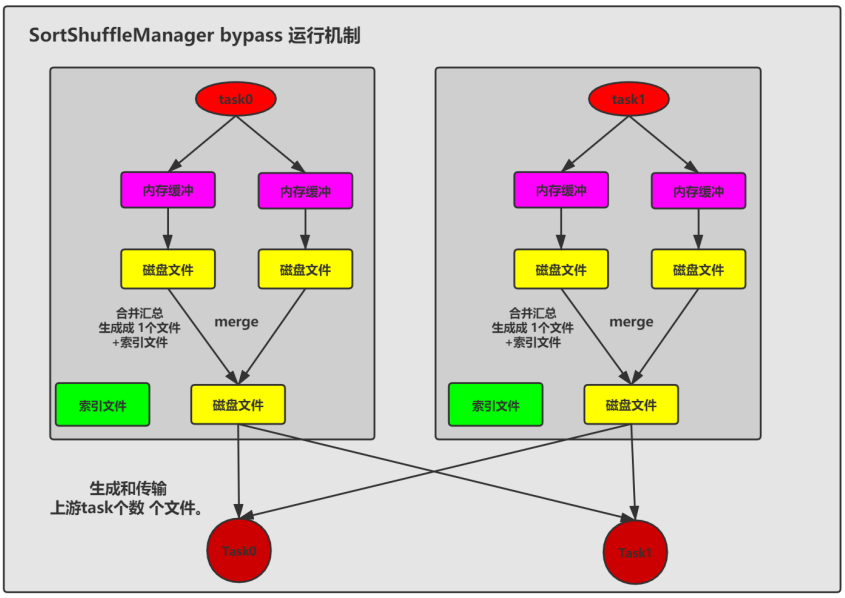
在我们使用 Spark-2.x 版本的 Shuffle 机制，只有 SortShuffleManager 这一种。但是这一种 Shuffle 有两种可能的情况：

```
1、普通运行机制，其实跟mapreduce的shuffle机制一样
2、bypass机制，最大的有点，就是，如果满足了对应的一些条件之后，就会执行优化，从而能够提高shuffle的执行效率，没有排序动作！
```


未经优化的HashShuffleManager
excutor0 with one core
excutor1 with one core
task0
task1
task2
task3
buffer 内存
buffer
block fike
task
buffer
reduce memory
excutor num = 2
excutor core =1
task num =4
per exutor task =2
reduce task num =3
all file=task num*reduce task num=12


经过优化以后的HashShufferManager
excutor0 with one core
excutor1 with one core
task0
task1
task2
task3
buffer 内存
buffer
block fike
task
buffer
reduce memory
excutor num = 2
excutor core =1
task num =4
per exutor task =2
reduce task num =3
all file=excutor num*reduce task num=2*3=6


SortShuffleManager 普通运行机制
task0
task1
内存
Map or Array
达到一定阀值
排序
分批落盘，每批1万个
内存缓冲
磁盘文件
合并汇总
生成成 1个文件
+索引文件
merge
索引文件
磁盘文件
生成和传输
上游task个数 个文件。




```
1、mapreduce          mapper Shuffle          reducer shuffle
   spark               shuffle write           shuffle read

2、启动 bypass 机制的两个条件：
    1、没有聚合操作
    2、bypass的参数：下游Task < 200
```

5.5. Spark 内存模型

Spark 作为一个基于内存的分布式计算引擎，其内存管理模块在整个系统中扮演着非常重要的角色。理解 Spark 内存管理的基本原理，有助于更好地开发 Spark 应用程序和进行性能调优。

在执行 Spark 的应用程序时，Spark 集群会启动 Driver 和 Executor 两种 JVM 进程。

```
1、Driver 为主控进程，负责创建Spark上下文对象SparkContext，提交Spark作业（Job），并将作业转化为计算任务（Task），在各个Executor进程间协调任务的调度。

2、Executor 负责在工作节点上执行具体的计算任务，并将结果返回给Driver，同时为需要持久化的RDD提供存储功能。

3、Driver 的默认内存大小是：512M，Executor 的默认内存大小：1G，YARN集群中的 container 默认内存也是 1G，但是这两个参数都是可以调整的。

4、作为一个 JVM 进程，Executor 的内存管理建立在 JVM 的内存管理之上，Spark 对 JVM 的堆内（On-heap）空间进行了更为详细的分配，以充分利用内存。堆内内存的大小，由 Spark 应用程序启动时的 --executor-memory 或 spark.executor.memory 参数配置。

5、Spark 对堆内内存的管理是一种逻辑上的规划式的管理，因为对象实例占用内存的申请和释放还是都由 JVM 完成，Spark 只能在申请后和释放前记录这些内存
```

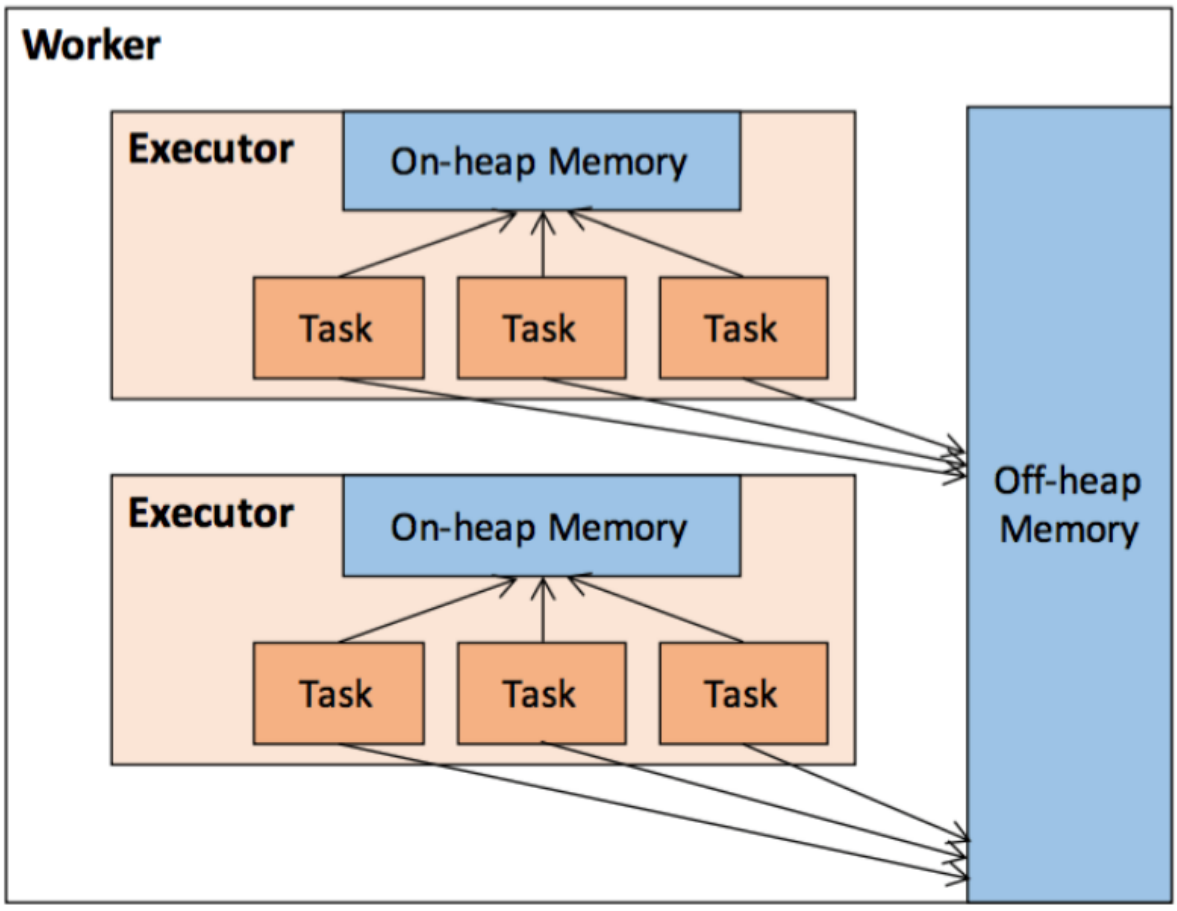
6、为了进一步优化内存的使用以及提高 Shuffle 时排序的效率，Spark 在 1.6 版本引入了堆外（Off-Heap）内存，使之可以直接在工作节点的系统内存中开辟空间，Spark 可以直接操作系统堆外内存，减少了不必要的内存开销，以及频繁的 GC 扫描和回收，提升了处理性能。在默认情况下堆外内存并不启用，可通过配置 spark.memory.offHeap.enabled 参数启用，并由 spark.memory.offHeap.size 参数设定堆外空间的大小。

7、Spark 为存储内存和执行内存的管理提供了统一的接口--MemoryManager，同一个 Executor 内的任务都调用这个接口的方法来申请或释放内存。

8、MemoryManager 的具体实现上，Spark 1.6之后默认为统一管理（UnifiedMemoryManager）方式，1.6之前采用的静态管理（StaticMemoryManager）方式仍被保留，可通过配置 spark.memory.useLegacyMode 参数启用。

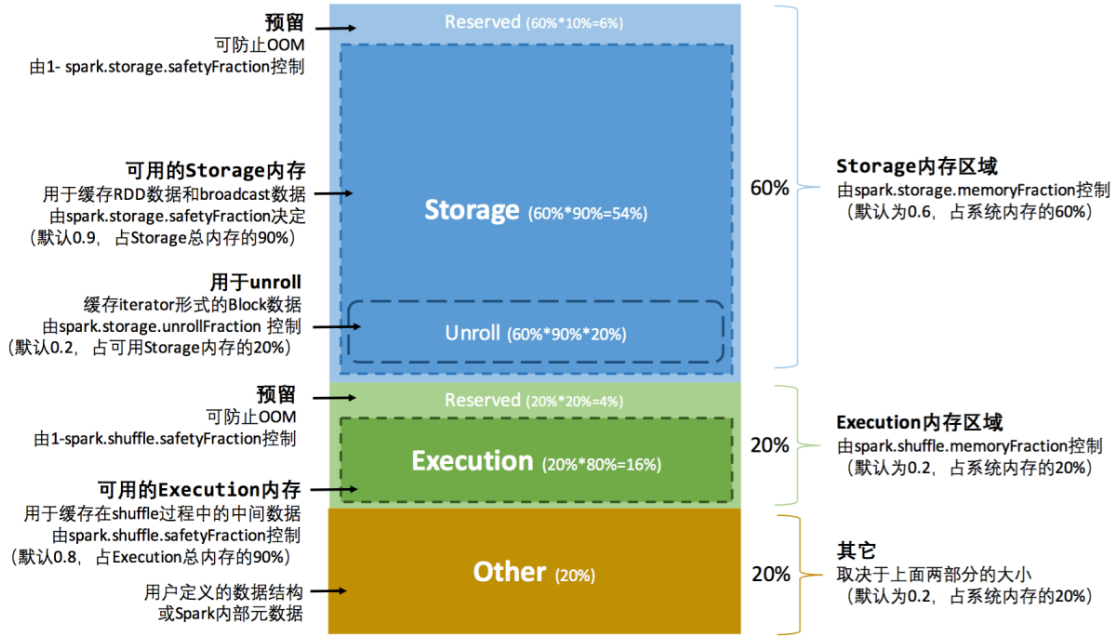
9、不管是堆内内存，还是堆外内存，都会被划为两块不同的功能区域，一个叫做执行内存，一个叫做存储内存。

关于堆内内存和堆外内存的物理状态图解：



5.5.1. 静态内存管理

在 Spark 最初采用的静态内存管理机制下，存储内存、执行内存和其他内存的大小在Spark应用程序运行期间均为固定的，但用户可以应用程序启动前进行配置，堆内内存的分配如下图所示：



JVM On-heap 内存

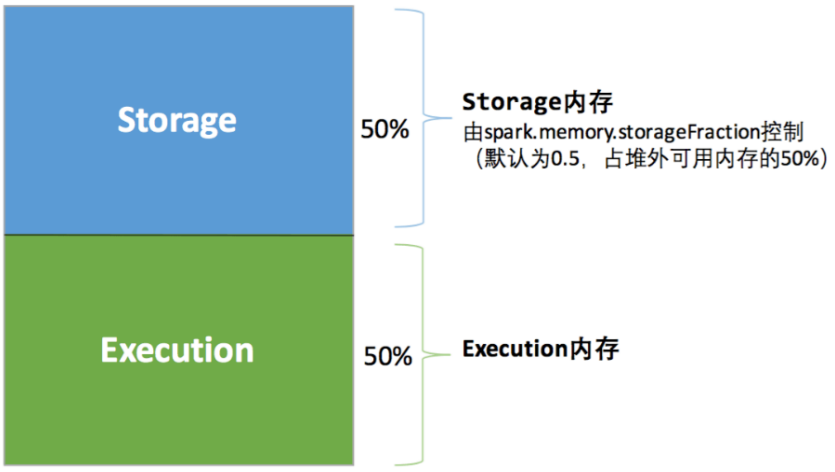
通过上图可以看出：可以看到，可用的堆内内存的大小需要按照下面的方式计算：

```
可用的存储内存 = systemMaxMemory * spark.storage.memoryFraction *
spark.storage.safetyFraction = 43.2%

可用的执行内存 = systemMaxMemory * spark.shuffle.memoryFraction *
spark.shuffle.safetyFraction = 16%

其中systemMaxMemory取决于当前JVM堆内内存的大小，最后可用的执行内存或者存储内存要在此基础上与
各自的memoryFraction参数和safetyFraction参数相乘得出
```

上述计算公式中的两个 safetyFraction 参数，其意义在于在逻辑上预留出 1-safetyFraction 这么一块保险区域，降低因实际内存超出当前预设范围而导致 OOM 的风险（Spark内存管理是一种规划式的管理，对于非序列化对象的内存采样估算会产生误差）。值得注意的是，这个预留的保险区域仅仅是一种逻辑上的规划，在具体使用时 Spark 并没有区别对待，和"其它内存"一样交给了JVM去管理。

可用的执行内存和存储内存占用的空间大小直接由参数 spark.memory.storageFraction 决定，由于堆外内存占用的空间可以被精确计算，所以无需再设定保险区域。

堆外的空间分配较为简单，只有存储内存和执行内存，如下图所示：



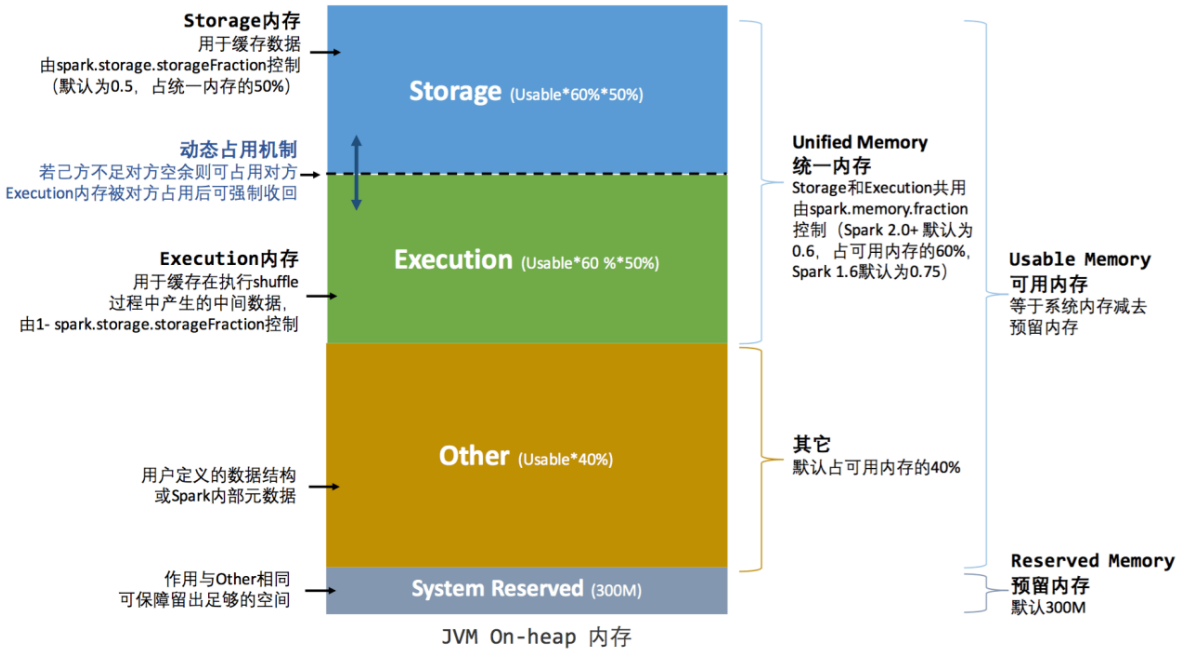
Off-heap内存

静态内存管理机制实现起来较为简单，但如果用户不熟悉 Spark 的存储机制，或没有根据具体的数据规模和计算任务或做相应的配置，很容易造成"一半海水，一半火焰"的局面，即存储内存和执行内存中的一方剩余大量的空间，而另一方却早早被占满，不得不淘汰或移出旧的内容以存储新的内容。由于新的内存管理机制的出现，这种方式目前已经很少有开发者使用，出于兼容旧版本的应用程序的目的，Spark 仍然保留了它的实现。

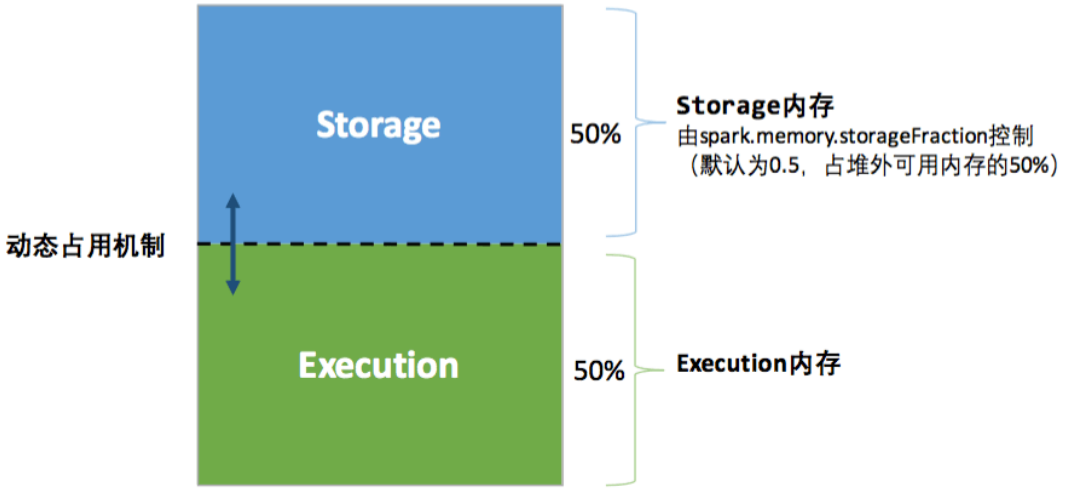
5.5.2. 统一内存管理

Spark-1.6 之后引入的统一内存管理机制，与静态内存管理的区别在于存储内存和执行内存共享同一块空间，可以动态占用对方的空闲区域。默认情况下，Spark 仅仅使用了堆内内存。Executor 端的堆内内存区域大致可以分为以下四大块：

```
1、Execution 内存：主要用于存放 Shuffle、Join、Sort、Aggregation 等计算过程中的临时数据
2、Storage 内存：主要用于存储 spark 的 cache 数据，例如RDD的缓存、unroll数据；
3、用户内存（User Memory）：主要用于存储 RDD 转换操作所需要的数据，例如 RDD 依赖等信息。
4、预留内存（System Reserved Memory）：系统预留内存，会用来存储Spark内部对象。
```



JVM On-heap 内存

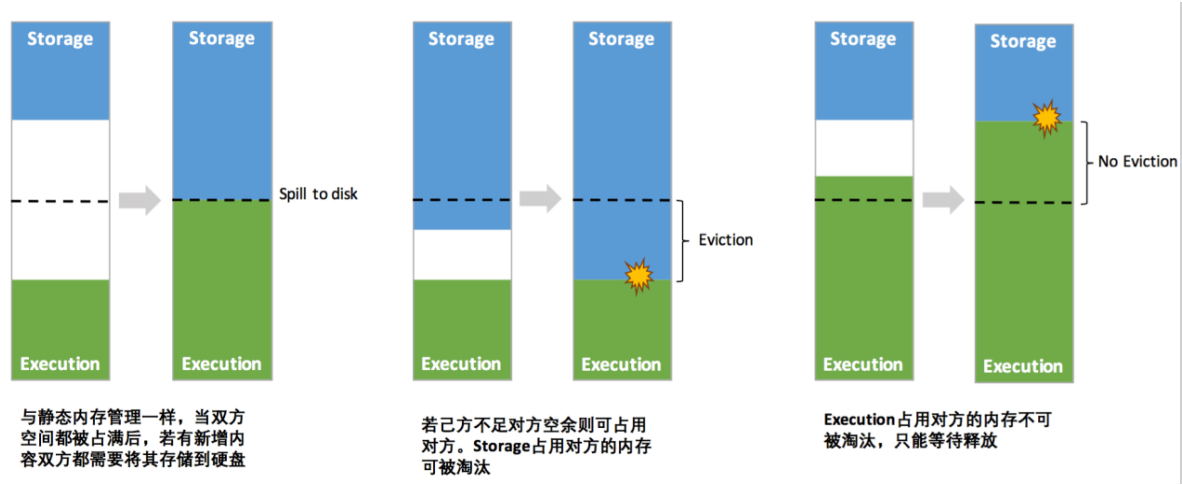


Off-heap内存

其中最重要的优化在于动态占用机制，其规则如下：

1、设定基本的存储内存和执行内存区域（spark.memory.storageFraction参数），该设定确定了双方各自拥有的空间的范围
2、双方的空间都不足时，则存储到硬盘；若己方空间不足而对方空余时，可借用对方的空间;（存储空间不足是指不足以放下一个完整的Block）
3、执行内存的空间被对方占用后，可让对方将占用的部分转存到硬盘，然后"归还"借用的空间
4、存储内存的空间被对方占用后，无法让对方"归还"，因为需要考虑Shuffle过程中的很多因素，实现起来较为复杂

动态占用机制如下：



凭借统一内存管理机制，Spark 在一定程度上提高了堆内和堆外内存资源的利用率，降低了开发者维护 Spark 内存的难度，但并不意味着开发者可以高枕无忧。譬如，所以如果存储内存的空间太大或者说缓存的数据过多，反而会导致频繁的全量垃圾回收，降低任务执行时的性能，因为缓存的 RDD 数据通常都是长期驻留内存的。所以要想充分发挥 Spark 的性能，需要开发者进一步了解存储内存和执行内存各自的管理方式和实现原理。

6. 本次课程总结

本次课程的内容，其实很多很多，整体来说，讲解了以下几方面的知识：

1、Spark DAG 引擎剖析
2、Spark 核心功能和架构设计
3、Spark 运行机制和任务执行流程详解
4、Spark Shuffle详解
5、Spark 内存模型

首先告诉你，一个优秀的 分布式计算调度引擎，该怎样去设计，然后当我们真正去设计一个分布式计算引擎的时候，需要那些功能支撑。然后根据这些基础功能去提供一些角色来完成相应的功能，规范好整个应用程序的执行机制。

另外，在理解 Spark 执行引擎的过程中，Shuffle 和 内存模型，是 Spark 的两块比较重要的功能，都跟大家在企业生产环境中，如何编写出高效率低资源消耗的 Spark 应用程序有很大的关系。

通过本次课程，希望大家能够了解到，对于要设计一款 分布式计算应用程序执行引擎，我们可以从 Spark 中，吸收那些有用的知识。

7. 本次课程作业

使用 Spark 的 RPC 框架实现一款 C/S 架构的聊天应用程序。

要求：

1、该应用应用程序，有最基本的服务端程序和客户端程序
2、首先启动服务端，保证服务端一直运行
3、然后启动客户端，客户端向服务端注册
4、再启动其他多个客户端，待启动的客户端注册之后，和其他客户端（当然，当前客户端必须具备感知其他客户端存在的功能，然后选择通信对象）进行通信。
5、客户端关闭，其他客户端收到通知。